

A New Algorithm for the Evaluation of Generalized B-splines

Ian D. Henriksen^a, Emily J. Evans^{a,*}, D. C. Thomas^b

^a*Department of Mathematics, Brigham Young University, Provo, Utah 84602, USA*

^b*Isogeometrx, Mapleton, Utah 84664, USA*

Abstract

In this paper we present a method for direct evaluation of generalized B-splines (GB-splines) via the local representation of these curves as piecewise functions. To accomplish this we introduce a local structure that makes GB-spline curves more amenable to the techniques used in constructing bases of higher degree. This basis is used to perform direct computation of piecewise representation of GB-spline bases and curves.

Keywords: GB-splines

1. Introduction

In computer aided geometric design (CAGD) the use of non-uniform rational B-splines (NURBS) as the basis for design is prevalent. The use of NURBS as the basis for geometric design is not without issues, however. First, NURBS cannot represent certain transcendental curves, many of which such as the helix and cycloid are used in design. Second, NURBS requires the use of weights to describe certain curves, the selection of which have no geometric meaning. Lastly the parametrization of conic sections does not correspond to the natural arc-length parametrization. Much research has been done in the computer aided design community to develop alternative technologies to the standard NURBS technology. Generalized B-splines (GB-splines) are one such technology that has received increased attention in recent years.

GB-splines are a generalization of B-splines that resolve some of the fundamental problems with the use of NURBS. Rather than spanning the spaces of piecewise polynomials spanned by traditional B-spline curves, on each interval $[t_i, t_{i+1})$ in the given knot vector T , they span the spaces $\{1, t, \dots, t^{p-2}, u_i^{[p-1]}, v_i^{[p-1]}\}$ where $u_i^{[p-1]}$ and $v_i^{[p-1]}$ are $p-1^{th}$ integrals of arbitrary functions forming a Chebyshev space over $[t_i, t_{i+1}]$. Because of their ability to span more general classes of functions, GB-splines allow exact representation of polynomial curves, helices and conic sections using control point representations that are intuitive and natural to designers [1]. GB-splines possess all of the fundamental properties of B-splines and NURBS that are important for

*Corresponding author

Email address: ejevans@mathematics.byu.edu (Emily J. Evans)

design and analysis such as local linear independence, degree-elevation and partition of unity. In addition to the geometric advantages of using GB-splines over NURBS, GB-splines also behave similarly to B-splines with respect to differentiation and integration. This similarity in behavior is especially beneficial when relevant properties of the continuous problem must be transferred to the discrete problem [2].

In 1999 Ksasov and Sattayatham [3] demonstrated a variety of the properties of GB-splines. In 2005 Costantini et al. [4] studied generalized Bernstein bases of this form in greater detail. In 2008, Wang et al. [5] introduced unified extended splines (UE-splines), a subclass of GB-splines and demonstrated that this new class of splines contains several other classes of generalized splines. In 2011, Manni et al. [6] proposed that GB-splines be used for isogeometric analysis. In [7], Romani successfully applied the techniques from [8] to form subdivision methods that allow for the approximation of UE-splines via a limit of control meshes successively refined by a non-stationary subdivision scheme. In [9], quasi-interpolation was performed in isogeometric analysis using GB-splines. Finally, GB-splines have also been used in the context of T-meshes [10].

From the usual recursive definition of GB-splines the only effective means of evaluating GB-splines is either through recursive numeric integration, or through symbolic computation of indefinite integrals. Recursive numeric integration is very costly for all but the lowest degrees of splines, whereas symbolic computation, while effective, can be unwieldy for numeric computation. In order to address these difficulties, we present a more direct method of computing values on GB-spline curves, using local representations.

1.1. Structure and content of the paper

In Section 2 the GB-splines are reviewed and appropriate notational conventions are introduced. An algorithm for their direct evaluation is introduced in Section 3. In Section 4 we draw conclusions.

2. A review of generalized B-splines

Generalized B-splines (GB-splines) were introduced in [5], and span spaces of the form $\{1, t, \dots, t^{p-2}, u(t), v(t)\}$ where u and v are more general functions defined over each interval in a knot vector. GB-splines retain most of the desirable properties of B-splines and unify a variety of other spline types such as UE-splines, trigonometric splines, exponential splines, etc. The primary advantages of GB-spline curves over traditional B-splines is that they allow for the exact representation of certain geometric curves and surfaces, like circles, hyperbolas, spheres, and hyperboloids, that cannot be well-represented by polynomial splines. Before formally defining a GB-spline we present some preliminary definitions.

Definition 2.1. A knot vector, T , is a nondecreasing vector of real numbers.

Definition 2.2. A set of ℓ linearly independent functions are said to form a Chebyshev space over an interval I if any nonzero function in their span has at most $\ell - 1$ roots in that interval.

Definition 2.3. Given a knot vector T , and functions u_i and v_i forming a Chebyshev space on each $[t_i, t_{i+1}]$ of nonzero length such that $u(0) = v(1) = 1$ and $v(0) = u(1) = 0$, we will refer to the sets of functions u_i and v_i as the knot functions over T .

With these definitions we are prepared to define a GB-spline.

Definition 2.4. Given a degree p and a knot vector T of length m with corresponding knot functions u_i and v_i . Define the i^{th} GB-spline basis function of degree p , denoted by N_i^p as follows:

Define the degree 1 GB-spline basis function as:

$$N_i^1(t) = \begin{cases} u_i(t) & t \in [t_i, t_{i+1}) \\ v_{i+1}(t) & t \in [t_{i+1}, t_{i+2}] \\ 0 & \text{otherwise} \end{cases},$$

For $p \geq 1$ define

$$\delta_i^p = \int_{t_i}^{t_{i+p+1}} N_i^p(s) ds.$$

For $p \geq 1$ define $\Phi_i^p(t)$ as

$$\Phi_i^p(t) = \begin{cases} \frac{\int_{t_i}^t N_i^p(s) ds}{\delta_i^p} & \text{if } \delta_i^p \neq 0, \\ 0 & \text{if } \delta_i^p = 0 \text{ and } t < t_{i+p+1}, \\ 1 & \text{if } \delta_i^p = 0 \text{ and } t \geq t_{i+p+1}. \end{cases}$$

For $p > 1$, define

$$N_i^p(t) = \Phi_i^{p-1}(t) - \Phi_{i+1}^{p-1}(t).$$

In addition, if $t_{m-p-1} = \dots = t_{m-1}$ and $t_{m-p-2} \neq t_{m-p-1}$ (that is, if the last p knots are repeated, and the last basis function is nonzero), define $N_{m-p-2}^p(t_{m-1}) = 1$.

Definition 2.5. Given a degree $p > 1$, and a knot vector T of length m with a corresponding set of knot functions, a degree $p > 1$, and $m - p - 1$ control points a_i , define the corresponding GB-spline curve $f(t)$ as

$$f(t) = \sum_{i=0}^{m-p-2} a_i N_i^p(t)$$

for $t \in [t_p, t_{m-p-1}]$.

Remark 2.6. In these definitions, and in the algorithms presented later in the manuscript we used zero-based indexing on the basis functions. That is to say instead of indexing the basis functions from $1, \dots, m$, we instead index the basis functions from $0, \dots, m - 1$.

Definition 2.7. Given a knot vector T with corresponding sets of knot functions u_i and v_i , define

$$V_i^p = \text{span} \left\{ 1, (t - t_i), \dots, (t - t_i)^{p-2}, u^{[p-1]}(t), v^{[p-1]}(t) \right\}$$

Where $u^{[p-1]}$, and $v^{[p-1]}$ are the $(p-1)^{th}$ indefinite integrals of u and v respectively.

GB-splines have the following important properties:

- B-splines are GB-splines [5].
- The support of N_i^p is zero outside the interval $[t_i, t_{i+p+1}]$.
- Partition of unity.
- GB-spline basis functions are linearly independent and positive on the interior of their supports [3, 6].
- GB-spline curves are variation diminishing [3, 6].
- A GB-spline over an open knot vector T with no degenerate basis functions in the corresponding spline basis interpolates its endpoints.
- A GB-spline curve $f(t)$ of degree p over a knot vector T with knot functions u_i and v_i has the following properties:
 - The GB-spline basis restricted to each interval lies in V_i^p .
 - Each GB-spline is C^{p-k} at each of the knots in the knot interval where k is the number of times a knot is repeated.
 - Each GB-spline is at least C^{p+r-1} for each point t not in its knot vector, where r is the minimum continuity of the knot functions over the interval in the knot vector that contains t .
- Where it exists, the derivative of a GB-spline basis function is given by

$$(N_i^p)'(t) = \frac{N_i^{p-1}(t)}{\delta_i^{p-1}} - \frac{N_{i+1}^{p-1}(t)}{\delta_{i+1}^{p-1}}.$$

- Where it exists, the derivative of a GB-spline curve $f(t)$ with control points a_0, \dots, a_n is given by

$$\sum_{i=0}^{n+1} a_i \frac{N_i^{p-1}(t)}{\delta_i^{p-1}} - a_{i+1} \frac{N_{i+1}^{p-1}(t)}{\delta_{i+1}^{p-1}}$$

with a_{-1} and a_{n+1} defined to be 0.

3. An algorithm for evaluation of GB-splines

Although it is clear from Definition 2.4 why many of the properties of GB-spline curves are true, it does not provide for a simple means of evaluation. From the definition the only effective means of evaluating GB-splines are either recursive numeric integration, or symbolic computation of indefinite integrals. Recursive numeric integration is very costly for all but the lowest degrees of splines, whereas symbolic computation, while effective, can be unwieldy for numeric computation. In order to address these difficulties, we present a more direct method of computing values on GB-spline curves.

Given that each basis function lies in the space V_i^p , we may introduce a local representation of each basis function in terms of the functions spanning the space V_i^p . Since the recursive integrals must be computed, we would like for these local representations to be more amenable to integration. These requirements introduce several possible choices for the local representations of the splines, but we will use the local representation given by $u_i^{[p-1]}$, $v_i^{[p-1]}$, and an additional polynomial term of degree $p-2$. By virtue of the linear independence of $u_i^{[p-1]}$ and $v_i^{[p-1]}$ from all other polynomial terms this is a valid choice of basis.

3.1. Local Representations: Knot Functions and Polynomials

Definition 3.1. Given a degree p and a knot vector T of length m with corresponding knot functions u_i and v_i , and since N_i^p lies in the space V_i^p , we can represent N_i^p on the j^{th} interval in T as:

$$N_i^p(t) = P_{i,j}^p(t) + a_{i,j}^p u_j^{[p-1]}(t) + b_{i,j}^p v_j^{[p-1]}(t)$$

where $P_{i,j}^p$ is a polynomial term and $a_{i,j}^p$ and $b_{i,j}^p$ are constants. The recurrence stated in Definition 2.4 can be written as:

$$a_{i,j}^p = \frac{a_{i,j}^{p-1}}{\delta_i^{p-1}} - \frac{a_{i+1,j}^{p-1}}{\delta_{i+1}^{p-1}},$$

$$b_{i,j}^p = \frac{b_{i,j}^{p-1}}{\delta_i^{p-1}} - \frac{b_{i+1,j}^{p-1}}{\delta_{i+1}^{p-1}},$$

and

$$P_{i,j}^p(t) = N_i^p(t_j) + \frac{1}{\delta_i^{p-1}} \left(\int_{t_j}^t P_{i,j}^{p-1}(s) ds - a_{i,j}^{p-1} u_j^{[p-1]}(t_j) - b_{i,j}^{p-1} v_j^{[p-1]}(t_j) \right) - \frac{1}{\delta_{i+1}^{p-1}} \left(\int_{t_j}^t P_{i+1,j}^{p-1}(s) ds - a_{i+1,j}^{p-1} u_j^{[p-1]}(t_j) - b_{i+1,j}^{p-1} v_j^{[p-1]}(t_j) \right).$$

With the additional stipulation that if N_i^{p-1} is identically zero and the interval $[t_{i+p}, t_{i+p+1})$ is empty, then an additional 1 is added to $P_{i,j}$ to account for the modified

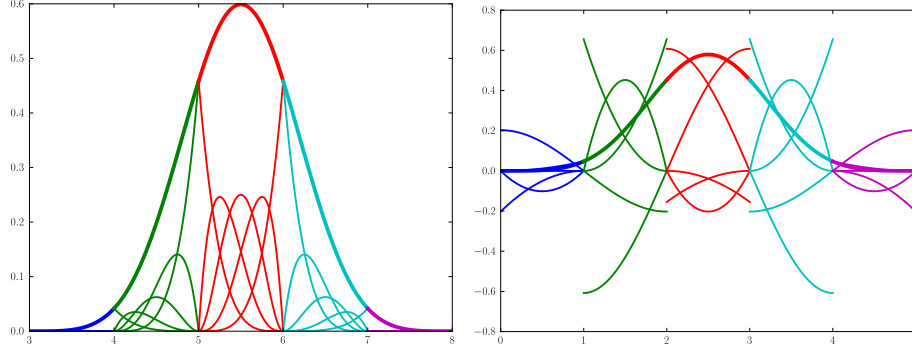


Figure 1: The local polynomial representation for a uniform B-spline basis function of degree 4 compared with the local representation for a uniform GB-spline function of degree 4 defined using trigonometric knot functions.

treatment of basis functions that are identically 0 in Definition 2.4. As before, we also require that, if the last basis function is discontinuous at the end of the $m - p - 1$ knot, that it must take a value of 1 at that knot.

Remark 3.2. For ease of notation later in the manuscript we name the coefficients $a_{i,j}^p$ and $b_{i,j}^p$ in Definition 3.1 as the general function coefficients.

The recurrence relation outlined in Definition 3.1 is not as easy to implement as De Boor's recurrence, however it does make it so that the evaluation of GB-spline curves is no longer tied to symbolic integrals or recursive quadrature. It makes it clear that the values of N_i^p on the interval $[t_j, t_{j+1})$ depend only on $N_i^p(t_j)$, the values of $u_j^{[p-1]}$ and $v_j^{[p-1]}$ at t and t_j , and the full set of coefficients for N_i^{p-1} and N_{i+1}^{p-1} . These dependencies can be stated more explicitly. To evaluate N_i^p at time $t \in [t_j, t_{j+1})$, it is necessary to know the values of the following function values:

- $u_j^{[p-1]}$ and $v_j^{[p-1]}$ at t ;
- the values of all the different $u_j, u_j^{[1]}, \dots, u_j^{[p-2]}$ and $v_j, v_j^{[1]}, \dots, v_j^{[p-2]}$ at t_j and $t_j + 1$ for each index j corresponding to an interval in the support of N_i^p ;
- the values of $u_j^{[p-1]}$ and $v_j^{[p-1]}$ if j is an index corresponding to an interval in the support of N_i^p at t_j if $t_j < t$ and at t_{j+1} when $t_{j+1} < t$.

3.2. An algorithm for evaluation of GB-splines

To construct a direct algorithm for the evaluation of arbitrary GB-spline curves, we must first determine how best to handle the dependencies between the intervals. Given that the representation of N_i^p on $[t_j, t_{j+1})$ depends on the representations of N_i^{p-1} and N_{i+1}^{p-1} over their supports and the representations of N_i^p over the intervals of its support that lie to the left of $[t_j, t_{j+1})$, it is natural to construct the set of basis

functions of each degree using the set of basis functions of the degree one less than the one being computed. The computation most naturally runs from left to right along each basis function. Given this structure, the algorithm should operate roughly as follows:

- Initialize a list of basis functions using the known values for the degree 1 case.
- For each degree from 2 to the desired degree p , do the following:
 - Integrate each polynomial term in the basis.
 - Use the polynomial terms, the general function coefficients, and the values of the indefinite integrals of the knot functions at the points in the knot vector to compute the definite integral of each basis function over its support.
 - Divide the indefinite integrals of the polynomial terms by the definite integral of the basis function they represent.
 - Divide the general function coefficients by the definite integral of the basis function they represent.
 - Compute the differences between the scaled general function coefficients for basis functions whose indices differ by 1.
 - Compute the differences between the scaled polynomials for basis functions whose indices differ by 1, adding the constant terms from the general function part to the polynomial.
 - Use the values of these differences to add the value of each basis function over each interval to its polynomial term over each interval.
 - Store these differences between the polynomial and general function terms as the new set of basis functions.

In practice, the functions that we desire to include in the span of the spline basis may not always satisfy the constraints on the values of the knot functions at the endpoints of each interval. This can be resolved by taking linear combinations of the original functions on each interval so that the endpoint constraints are satisfied. This can be taken care of as a part of the algorithm for constructing a basis by taking the needed linear combinations of the integral terms given as input and then, once the local representations of the spline basis have been computed, changing the representations so that they are given with respect to the original functions rather than the computed linear combinations. In order to ensure the desired properties of a spline basis, the functions used to create the knot functions must still form a Chebyshev space over each corresponding nondegenerate interval in the knot vector. The matrix

$$\begin{bmatrix} u_i(t_i) & v_i(t_i) \\ u_i(t_{i+1}) & v_i(t_{i+1}) \end{bmatrix}$$

must also be invertible (and sufficiently well-conditioned) so that the needed linear combinations can be computed.

In addition, the spline basis constructed will span $u^{[p-1]}$ and $v^{[p-1]}$, not u and v . It is often desirable to construct a basis that spans C^{p-1} functions \tilde{u} and \tilde{v} instead. To handle that properly, we need only begin the iteration with the knot functions

$\tilde{u}^{(p-1)}$ and $\tilde{v}^{(p-1)}$, noting that, after the corresponding numbers of integrals have all been taken, the spline basis will span the desired functions. To use this approach, it is necessary that there exist linear combinations of $\tilde{u}^{(p-1)}$ and $\tilde{v}^{(p-1)}$ that satisfy the constraints that would normally be imposed on u and v .

The recurrence outlined in Definition 3.1 is restated as an algorithm in Algorithm 1. For convenience, each basis is stored as two arrays, the first containing the polynomial terms corresponding to each interval within the support of each basis function and the second containing the corresponding abstract function terms, that is the terms for u and v . Given that the support of each basis function is known, we only include the representation of each basis function on an interval where it will be nonzero. This shifts the indices for the representation of each basis function, but the structure of the iteration is essentially the same. The algorithms here will be presented in a form that is independent of the polynomial basis used.

For practical use it is also helpful to follow the convention that the knot functions and polynomials corresponding to each interval are defined on the interval $[0, t_{i+1} - t_i]$ and that $t - t_i$ is used as an argument rather than t itself. This makes it so that, for any given polynomial representation requiring boundaries of definition (Bernstein polynomials, Chebyshev polynomials, etc.), only the lengths of each interval must be passed to the polynomial integration and evaluation routines.

An important consequence of using the piecewise representations for these basis functions is that, once a piecewise representation for a spline curve is created, the only remaining costs of evaluating the function at any given point come from identifying which interval in the knot vector contains the given parameter value, evaluating a polynomial term, and evaluating the terms $u_i^{[p-1]}$ and $v_i^{[p-1]}$. No further recursion or integration is needed.

The algorithms will be presented in vectorized form with a particular emphasis on clarity. A variety of other small optimizations could be added to further remove redundant computations; however the presentation here is meant primarily to provide a clear explanation of the algorithm. It presents a relatively efficient version of the algorithm, but, for simplicity, redundant computations have not been completely removed.

For clarity within the algorithm and its helper routines, we will now introduce the many variables used throughout this algorithm and its corresponding helper routines. Throughout the code for this algorithm, the following variables will be used:

- T is a 1-dimensional array containing the knot vector.
- $Tlens$ is a 1-dimensional array containing the lengths of the intervals between the knots values in T .
- $Tvals$ an array shaped like $Tlens$ containing the lengths of each interval. $Tvals$ is indexed first by basis function, then by interval within the support of a given basis function.
- p is the degree of the desired basis, or of the spline to be evaluated.
- n is the number of basis functions in a given basis.

- *ints* is a 4-dimensional array of shape $(p, \text{len}(t) - 1, 2, 2)$ containing the values of the indefinite integrals of the knot functions at the endpoints of each interval. The p^{th} integrals are indexed in ascending order along the first axis. The different intervals within the knot vector are indexed along the second axis. The different endpoints of each interval are indexed along the third axis. The different knot functions (u and v) are indexed along the last axis with u first.
- *wints* is a 3-dimensional view into *ints* corresponding to the integral values of a given degree, indexed first by basis function, then by interval within the support of each basis function, then by endpoint, then by the different knot functions.
- *polys* is an array containing the coefficients for the polynomial parts of the basis functions in a given basis. Basis functions are indexed along the first axis and intervals within the support of each basis function are indexed along the second axis. Here we will assume that the polynomial term over the i 'th interval is stored in the form $p(t - t_i)$, that is, that the polynomial terms are translated so that the first value taken by the polynomial in each interval is the value of the polynomial at 0. This algorithm does not depend on the representation used to store the polynomial terms, but in most cases an array of shape $(n, p + 1, p - 1)$ containing only the necessary coefficients should suffice.
- *pints* is an array containing the integrals of all the polynomial terms in a given array *polys*. All the axes are indexed the same as the axes in *polys*. The shape will be the same except that the last axis will be one index longer than the last axis of *polys*.
- *genfunc* will be an array containing the coefficients for the general function terms of the basis functions in a given basis. Basis functions should be indexed along the first axis, intervals within the support of each basis function along the second axis, and the two general function coefficients along the third (with u first, then v). This array will have a shape of $(n, p + 1, 2)$.
- *scal* will be an array containing the necessary scaling matrices needed to scale *ints* to represent the scaled versions of u and v that satisfy the value constraints at their endpoints and also needed to scale the coefficients in *genfunc* so that they represent the basis functions in terms of the original u and v .
- *pos* will be an array of boolean values. The i 'th entry of *pos* will be true if $\delta_i^{d-1} \neq 0$.
- *deltas* will be an array containing the indefinite integrals of each basis function over its support.
- *consts* will be an array containing the constant terms to be added to the polynomial terms on each interval. It will be indexed first by basis function, then by interval within the support of each basis function. In the recurrence in Definition 3.1, these are the terms

$$-a_{i,j}^{d-2} u_j^{[d-2]}(t_j) - b_{i,j}^{d-2} v_j^{[d-2]}(t_j)$$

- *vals* will be a temporary array used to store outputs of various functions.
- *d* will be a looping variable used in the loop that constructs the basis of each degree from the basis of previous degree. Here *d* will be the degree of the basis being constructed. *dmin* will be equal to $d - 1$.

Algorithm 1 shows the primary routine used to compute the local representations of a given basis function. It contains calls to a variety of auxiliary routines, all of which will be explained here. Here we include the primary algorithm first so that the reader may understand the general flow of the algorithm and the proper place for each auxiliary routine before handling the many details that are taken care of in the auxiliary routines.

Algorithm 1 uses the following auxiliary routines:

- *Wrap*: A utility function used to convert between indexing by interval to indexing by basis function, then by interval within the support of each basis function.
- *MatMul*: A utility function that performs matrix multiplication.
- *MakeDegreeOne*: A function that initializes the coefficient arrays for a basis of degree 1.
- *ScaleKnotFuncs*: A function that computes the scaled *ints* and the corresponding array *invs* containing the scalings. This function is what changes all the basis functions to be represented in terms of the linear combinations of the functions used to create *invs* that satisfy the required constraints to be knot functions.
- *ScaleGenFuncCoefs*: A function that modifies *genfunc* in-place to change the coefficients to represent the basis functions in terms of the functions used to create *invs*.
- *PolyInt*: A function that, given an array of polynomial coefficients with the coefficients indexed along the last axis and another array containing the lengths of the intervals corresponding to each polynomial term, computes the indefinite integrals of all the polynomials.
- *PolyVal*: A function that, given an array of polynomial coefficients with the coefficients indexed along the last axis, and an array containing the lengths of the intervals corresponding to each polynomial term, evaluates each polynomial at the corresponding term in an array *vals*. This function is used only within other auxiliary routines.
- *IntegrateSupports*: A function that computes the integral of each basis function over its support
- *GenFuncInts*: A helper function called within *IntegrateSupports*. It computes the portion of the integral of each basis function that comes from the knot functions over each interval in its support.

Algorithm 1 Computing the local coefficients for a GB-spline basis

```
1: procedure BASISCOEFS( $T, ints, tol = 10^{-8}$ )
2:    $\triangleright$  Initialize  $p$ ,  $Tlens$ , and  $Tvals$  and coefficient arrays for a basis of degree 1.
3:    $p = \text{shape}(ints)[0]$ 
4:    $Tlens = T[1:] - T[: -1]$ 
5:    $Tvals = \text{Wrap}(Tlens, 2)$ 
6:    $polys, genfunc = \text{MakeDegreeOne}(\text{shape}(T)[0] - 2)$ 
7:    $\triangleright$  Take linear combinations of the functions with integrals in  $ints$  so that
8:    $\triangleright$  the resulting linear combinations satisfy constraints on knot functions.
9:    $ints, scal = \text{ScaleKnotFuncs}(ints, Tlens, tol)$ 
10:   $\triangleright$  Construct each successive set of local coefficients.
11:  for  $d = 2, d \leq p$  do
12:     $\triangleright$  Compute the indefinite integrals of all polynomial terms
13:     $pints = \text{PolyInt}(polys, Tvals)$ 
14:     $\triangleright$  Construct  $wints$  by wrapping the first axis of  $ints$  into two new axes.
15:     $wints = \text{Wrap}(ints[d - 1], d)$ 
16:     $\triangleright$  Integrate the current set of basis functions over their supports.
17:     $deltas, consts = \text{IntegrateSupports}(Tvals, pints, genfunc, wints)$ 
18:     $\triangleright$  Add constant terms from the general function integrals to the  $pints$ .
19:     $\text{OffsetConstants}(pints, consts)$ 
20:     $\triangleright$  Compute the indices of the basis functions that are identically 0.
21:     $pos = (T[d:] - T[: -d]) > tol$ 
22:     $\triangleright$  Take the deltas corresponding to positive basis functions.
23:     $\triangleright$  Also reshape the deltas for broadcasting with  $pints$  and  $genfunc$ .
24:     $deltas = deltas[pos, None, None]$ 
25:     $\triangleright$  Divide the terms in  $pints$  and  $genfunc$  by their corresponding entries in
     $deltas$ .
26:     $pints[pos] /= deltas$ 
27:     $genfunc[pos] /= deltas$ 
28:     $\triangleright$  Take the differences between neighboring terms in  $pints$  and  $genfunc$ .
29:     $polys, genfunc = \text{OffsetDifferences}(pints, genfunc)$ 
30:     $\triangleright$  Add ones where needed to account for the integral terms of 0-valued
31:     $\triangleright$  basis functions after the knot value where their support would end.
32:     $\text{AddOnes}(polys, pos)$ 
33:     $\triangleright$  Compute the set of  $Tvals$  for the next basis.
34:     $Tvals = \text{Wrap}(Tlens, d + 1)$ 
35:     $\triangleright$  Add in the constant terms that come from evaluating each basis function
36:     $\triangleright$  at the end of each interval within the knot vector.
37:     $\text{ConnectBoundaries}(polys, genfunc, wints, Tvals)$ 
38:  end for
39:   $\triangleright$  Scale the coefficients in  $genfunc$  so that the basis functions are represented
40:   $\triangleright$  in terms of the general function terms originally represented in  $ints$ .
41:   $\text{ScaleGenFuncCoefs}(\text{Wrap}(scal, p + 1), genfunc)$ 
42:  return  $polys, genfunc$ 
43: end procedure
```

- *OffsetConstants*: A function that modifies *pints* in-place to add in the constant terms that come from the general function integral. In the polynomial part of the recurrence from Definition 3.1, this accounts for the terms

$$-a_{i,j}^{p-1} u_j^{[p-1]}(t_j) - b_{i,j}^{p-1} v_j^{[p-1]}(t_j)$$

and

$$a_{i+1,j}^{p-1} u_j^{[p-1]}(t_j) - b_{i+1,j}^{p-1} v_j^{[p-1]}(t_j)$$

- *OffsetDifferences*: A function that takes the *pints* and *genfunc* (after each term has been divided by the corresponding δ_i terms) that correspond to a given basis and computes the differences between consecutive terms. This returns the differences between both the polynomial and general function terms. These terms account for all terms in the recurrence from Definition 3.1 with the exception of $N_i^p(t_j)$.
- *AddOnes*: A function that adds the one terms to *polys* that come from the handling of the integral terms from basis functions that are identically 0 as defined in Definition 2.4.
- *ConnectBoundaries*: A function that computes the term $N_i^p(t_j)$ for each interval where it is needed and adds it in place to *polys*.

3.3. Auxiliary Routines

In this section we discuss the implementations of the auxiliary routines in greater detail.

The helper routine *Wrap* is used to expand a given axis into two axes where each index of the first of the two new axes provides a moving window of the given width along the original axis that is being expanded. Within this algorithm, this function is used to convert between data that is indexed by interval within the knot vector to data that is indexed by basis function, then by interval within the support of each basis function.

The implementation for *MakeDegreeOne* should also be relatively simple. Recall that each degree 1 basis function has the form

$$\begin{cases} v_i(t) & x \in [t_i, t_{i+1}) \\ u_i(t) & t \in [t_{i+1}, t_{i+2}) \end{cases}$$

This means that this function should allocate *polys* as an empty array of shape $(n, 2, 0)$ and allocate *genfunc* as an array of 0's of shape $(n, 2, 2)$. Then it should fill *genfunc* with values such that $genfunc[i, j, k]$ is equal to 0 when $j = k$ and 1 otherwise. It should then return *polys* and *genfunc*.

ScaleKnotFuncs allows derivatives from functions that do not necessarily satisfy the constraints $u_i(t_i) = v_i(t_{i+1}) = 1$ and $u_i(t_{i+1}) = v_i(t_i) = 0$ for the integral values stored in *ints*. To do this, it must require that, for each nonempty interval $[t_i, t_{i+1})$, the matrix

$$A_i = \begin{bmatrix} u_i(t_i) & v_i(t_i) \\ u_i(t_{i+1}) & v_i(t_{i+1}) \end{bmatrix}$$

be invertible and reasonably well-conditioned. It is still required that u_i and v_i form a Chebyshev space.

Since the algorithm for constructing the basis coefficients relies on each u_i and v_i satisfying the constraints on its values at the endpoints of each interval in the knot vector, we must compute the linear combinations of u and v that satisfy the value constraints at each endpoint. Since matrix multiplication can be seen as using the columns of the matrix on the right as coefficients for linear combinations of the columns of the matrix on the left, we see that the matrix B_i with the desired coefficients for the linear combinations must satisfy the equation $A_i B_i = I$, so $B_i = A_i^{-1}$. Since it is only necessary to invert matrices of size 2×2 , for simplicity we will content ourselves with using a direct matrix inverse to compute the new derivatives, though other methods could be used to compute the desired derivative and integral values.

Now, given the matrices B_i , we must now use the coefficients for the desired linear combinations stored as columns of B_i to compute the corresponding integral terms. Using similar reasoning as before, taking the needed linear combinations of the integral terms stored in *ints* corresponds to right-multiplication of each 2×2 matrix corresponding to a given degree and interval by the matrix B_i corresponding to that interval. This routine must return both the new scaled version of *ints* and the corresponding matrices B_i (these are the i 'th entry along the first axis of *ints*) because the matrices B_i must be used again later to write the computed coefficients for the general functions in terms of the original u and v rather than their scaled linear combinations. The internal workings of this auxiliary routine are outlined in Algorithm 2.

Algorithm 2 Take linear combinations of the input knot functions such that the desired linear combinations will satisfy the constraints $u_i(t_i) = v_i(t_{i+1}) = 1$ and $u_i(t_{i+1}) = v_i(t_i) = 0$. Return the corresponding integral terms of these linear combinations and the coefficients for the linear combinations over each interval.

```

1: procedure SCALEKNOTFUNCS(ints, Tlens, tol =  $1E-8$ )
2:   ▷ Copy ints so it can be modified in-place without modifying the input array.
3:   ints = copy(ints)
4:   ▷ Get a boolean array showing where the the lengths of the intervals are
   nonzero.
5:   pos = Tlens > tol
6:   ▷ Compute the coefficients of the needed linear combinations.
7:   invs = array of 0's of shape shape(ints) [1 :]
8:   invs [pos] = inv(ints [0, pos])
9:   ▷ Perform matrix multiplication of each set of coefficients for each interval
   and degree
10:  ▷ by the corresponding scaling matrix for each interval.
11:  ints [:] = MatMul(ints, invs)
12:  return ints, invs
13: end procedure

```

Once the main loop in Algorithm 1 is finished, the computed general function coefficients must be changed to represent each basis function in terms of the original

knot functions rather than the chosen linear combinations of them. This is equivalent to left-multiplying the set of coefficients for each interval by the matrix B_i (as in the explanation for *ScaleKnotFuncs*). This process is shown in Algorithm 3.

Algorithm 3 Perform a change of basis on the general function coefficients so that the general function coefficients used to represent the basis correspond to the functions originally used to form the array *ints* of integral values.

```

1: procedure SCALEGENFUNCCOEFS(invs, genfunc)
2:   return MatMul (invs, genfunc [..., None]) [..., 0]
3: end procedure

```

The auxiliary routine *PolyInt* is dependent on the polynomial representation used. The array *Tvals* is used as an argument because the polynomial basis used could be defined over some given interval (as are the Bernstein, Chebyshev, and Legendre polynomials). For the power basis polynomials, that argument is not needed. As has already been mentioned, the interval lengths are all that is necessary since the polynomial and general function terms are all assumed to be shifted to be defined over an interval starting at 0.

The auxiliary routine *PolyVal* should be handled similarly as *PolyInt*. This routine is also dependent on the polynomial representation and is easily defined as using Horner's algorithm, the De Casteljau algorithm, Clenshaw's algorithm, or some other polynomial evaluation algorithm.

GenFuncInts is a function to compute the general function integrals

$$\int_{t_i}^{t_{i+1}} \left(a_{i,j}^{d-2} u_{i,j}^{[d-2]}(s) + b_{i,j}^{d-2} v_{i,j}^{[d-2]}(s) \right) ds$$

with the corresponding constant terms

$$-a_{i,j}^{d-2} u_j^{[d-2]}(t_j) - b_{i,j}^{d-2} v_j^{[d-2]}(t_j)$$

from the left endpoint of the integral. It is dependent on the representation used for the knot functions (we've only introduced using the knot functions themselves thus far). In the case that the knot functions themselves are used, Algorithm 4 shows how this can be done.

Algorithm 4 Integrate the general function part of each basis function over each interval in the support of that basis function.

```

1: procedure GENFUNCINT(genfunc, wints)
2:   consts = -sum (genfunc * wints[:, :, 0], axis = -1)
3:   vals = sum (genfunc * wints[:, :, 1], axis = -1) + consts
4:   return vals, consts
5: end procedure

```

IntegrateSupports is a function that evaluates the integrals of each basis function over its corresponding support. It should return both the desired indefinite integrals and

the constant terms (stored in variable *consts*) that come from the left bounds of each integral. This function is outlined in Algorithm 5.

Algorithm 5 Compute the definite integrals of each basis function over its support.

```

1: procedure INTEGRATESUPPORTS(Tvals, pints, genfunc, wints)
2:   ▷ wints and Tvals line the integral terms and the interval lengths up
3:   ▷ with their corresponding interval in each basis function.
4:   vals, consts = GenFuncInt(genfunc, wints)
5:   vals += PolyVal(pints, Tvals, Tvals)
6:   return sum(vals, axis = -1), consts
7: end procedure

```

OffsetConstants is another helper routine that interfaces with the polynomials and is dependent on how the polynomials are represented. It adds the terms stored in *consts* to the corresponding terms in *pints*. In the case of the power basis, Chebyshev basis, or Legendre basis, this can be done by adding each constant term to the term in the polynomial representation that represents constants. In the case of the Bernstein polynomials, since all the coefficients sum to 1, adding a constant is the same as adding a constant to each coefficient, so this operation can be performed by adding the constant term for each polynomial to all the coefficients for that polynomial.

OffsetDifferences is an auxiliary routine that takes care of differencing between the integrated terms from the previous basis function to form the differences over each interval that are needed to form the new basis. Once this function has been applied, the terms account for everything included in the recurrence in Definition 3.1 with the exception of the constant term for each interval that comes from evaluating each basis function on the right endpoint of the interval to the left of the current interval. This function also does not account for adding the ones to handle basis functions that are identically 0. The pseudocode for this algorithm is outlined in Algorithm 6.

Algorithm 6 Take the differences between the integral terms for the previous set of basis functions to start forming the new set of basis functions.

```

1: procedure OFFSETDIFFERENCES(pints, genfunc)
2:   n = shape(pints)[0]
3:   nints = shape(pints)[1]
4:   ▷ Allocate the arrays needed to store the coefficients for the new basis.
5:   npolys = new array of 0's of shape (n - 1, nints + 1, dmin)
6:   ngenfunc = new array of 0's of shape (n - 1, nints + 1, 2)
7:   ▷ Take the needed differences between the corresponding terms.
8:   npolys[:, :-1] += pints[:, -1]
9:   npolys[:, 1:] -= pints[1:]
10:  ngenfunc[:, :-1] += genfunc[:, -1]
11:  ngenfunc[:, 1:] -= genfunc[1:]
12:  return npolys, ngenfunc
13: end procedure

```

AddOnes is used to add the ones that come from the integral terms from Definition 2.4 that correspond to basis functions that are identically 0. We have separated it as an auxiliary routine because it both depends on the polynomial basis used. We add 1 only to the last interval of basis functions for which the first term of the recurrence from Definition 2.4 corresponds to a basis function that is identically 0. This is because, when constructing the basis functions of the next highest degree, the integral term corresponding to a basis function with index i appears only in the expressions for the basis functions at index $i - 1$ and i . Of those two basis functions, only the basis function at index i takes nonzero values on an interval that lies to the right of the support of the basis function that is identically 0. Once understood, this operation is very simple to perform, as can be seen in Algorithm 7, which demonstrates this auxiliary routine for polynomials represented in the power basis. Though this routine depends on the polynomial representation used, it is not necessary to pass *Tvals* since a constant term is the same for a polynomial represented over any interval.

Algorithm 7 Add ones where needed to account for the integral terms in Definition 2.4 that correspond to basis functions that are identically 0.

```

1: procedure ADDONES(polys, pos)
2:   ▷ Where the integral of the basis function of previous degree at the same
3:   ▷ index was 0, add 1 to the constant term of the last polynomial term.
4:   polys [ $\sim$  pos [ $:-1$ ],  $-1, -1$ ] += 1
5: end procedure

```

ConnectBoundaries is the last auxiliary routine needed to construct the new basis functions from the previous ones. In the recurrence in Definition 3.1, this function adds in the terms $N_i^p(t_j)$. This function effectively starts at the leftmost interval in the support of each basis function, computes the value of the basis function at the end of that interval, adds that constant term to the polynomial term of the basis function on the next interval, and continues until it has added the needed constant terms to every interval in the support of that basis function. This is done in a vectorized manner in Algorithm 8.

Algorithm 8 For each basis function, add in the constant terms $N_i^p(t_j)$ to each interval where they are needed.

```

1: procedure CONNECTBOUNDARIES(polys, genfunc, wints, Tvals)
2:   ▷ For each basis function, evaluate all but the leftmost polynomial term at the
   end
3:   ▷ of the interval where it is defined.
4:   vals = PolyVal(polys [ $:-1$ ], Tvals, Tvals [ $:-1$ ])
5:   ▷ Add in the corresponding values of the general functions.
6:   vals += sum(genfunc [ $:-1$ ] * wints [ $:-1, :, 1$ ], axis =  $-1$ )
7:   ▷ Add the needed constants to their corresponding polynomial terms.
8:   OffsetConstants(polys [ $:, 1 :$ ], cumsum(vals, axis =  $1$ ))
9: end procedure

```

4. Conclusion

In this manuscript we have presented an algorithm for the evaluation of GB-spline curves via their piecewise representation which is more direct than the recursive integral process given in the original definition for GB-splines. The new algorithm makes practical computation simpler and easier to implement. Moreover, using piecewise local representations make it so that the cost of evaluating a given spline curve is bound primarily by the costs of finding the portion of the knot vector in which a given point lies and evaluating the functions spanned by the spline basis. The computational routines here can also be used to work toward developing more efficient methods for the evaluation of specific classes of GB-spline curves. They provide working examples that can be used to further study possible ways to provide better evaluation routines or subdivision methods for specific classes of GB-spline curves.

The use of piecewise local representations for the evaluation of GB-spline curves motivates the use of these local representations for other operations. One such operation is GB-spline refinement which will be covered in a future paper by the authors. Also the local bases on each interval used to construct each spline curve share some of the useful properties of the bases used in [11]. There, the process of inserting knots to represent a given B-spline curve as a piecewise polynomial was used to develop a local element structure that can, in turn, be used in isogeometric analysis [6, 12]. The local representations introduced here can be used in a similar manner to provide element structures for Isogeometric Analysis.

References

- [1] E. Mainar, J. Pea, J. Snchez-Reyes, Shape preserving alternatives to the rational bezier model, *Computer Aided Geometric Design* 18 (1) (2001) 37 – 60. doi:http://dx.doi.org/10.1016/S0167-8396(01)00011-5.
- [2] A. Buffa, G. Sangalli, R. Vzquez, Isogeometric analysis in electromagnetics: B-splines approximation, *Computer Methods in Applied Mechanics and Engineering* 199 (17:20) (2010) 1143 – 1152. doi:http://dx.doi.org/10.1016/j.cma.2009.12.002.
- [3] B. I. Ksasov, P. Sattayatham, GB-splines of arbitrary order, *Journal of Computational and Applied Mathematics* 104 (1) (1999) 63 – 88. doi:10.1016/S0377-0427(98)00265-9.
- [4] P. Costantini, T. Lyche, C. Manni, On a class of weak Tchebycheff systems, *Numerische Mathematik* 101 (2) (2005) 333–354. doi:10.1007/s00211-005-0613-6.
- [5] G. Wang, M. Fang, Unified and extended form of three types of splines, *Journal of Computational and Applied Mathematics* 216 (2) (2008) 498 – 508. doi:10.1016/j.cam.2007.05.031.
- [6] C. Manni, F. Pelosi, M. L. Sampoli, Generalized B-splines as a tool in isogeometric analysis, *Computer Methods in Applied Mechanics and Engineering* 200 (58) (2011) 867 – 881. doi:10.1016/j.cma.2010.10.010.

- [7] L. Romani, From approximating subdivision schemes for exponential splines to high-performance interpolating algorithms, *Journal of Computational and Applied Mathematics* 224 (1) (2009) 383 – 396. doi:10.1016/j.cam.2008.05.013.
- [8] J. Warren, H. Weimer, *Subdivision Methods for Geometric Design: A Constructive Approach*, Morgan Kaufmann series in computer graphics and geometric modeling, Morgan Kaufmann, 2002.
- [9] P. Costantini, C. Manni, F. Pelosi, M. L. Sampoli, Quasi-interpolation in isogeometric analysis based on generalized B-splines, *Comput. Aided Geom. Design* 27 (8) (2010) 656–668. doi:10.1016/j.cagd.2010.07.004.
URL <http://dx.doi.org/10.1016/j.cagd.2010.07.004>
- [10] C. Bracco, F. Roman, Spaces of generalized splines over t-meshes, *Journal of Computational and Applied Mathematics* 294 (2016) 102 – 123. doi:<http://dx.doi.org/10.1016/j.cam.2015.08.006>.
- [11] M. J. Borden, M. A. Scott, J. A. Evans, T. J. R. Hughes, Isogeometric finite element data structures based on Bézier extraction of NURBS, *International Journal for Numerical Methods in Engineering* 87 (1-5) (2011) 15–47. doi:10.1002/nme.2968.
- [12] T. Hughes, J. Cottrell, Y. Bazilevs, Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement, *Computer Methods in Applied Mechanics and Engineering* 194 (3941) (2005) 4135 – 4195. doi:10.1016/j.cma.2004.10.008.